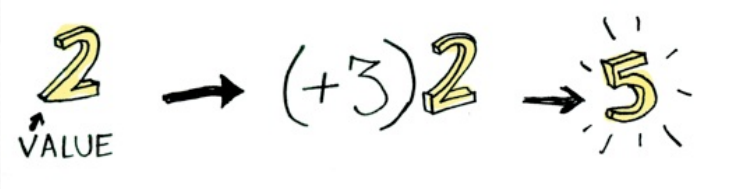
Contents

Functors, Applicatives, And Monads In Pictures

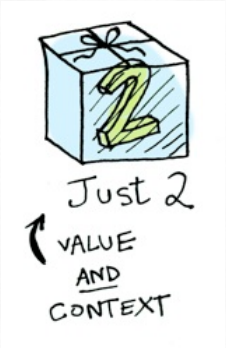
WRITTEN APRIL 14, 2013

Heres a simple value:

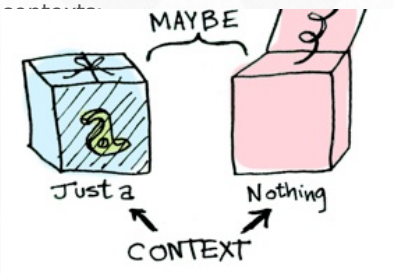
And we know how to apply a function to this value:



Simple enough. Lets extend this by saying that any value can be in a context. For now you can think of a context as a box that you can put a value in:



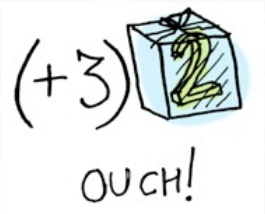
Now when you apply a function to this value, youll get different results **depending on the context**. This is the idea that Functors, Applicatives, Monads, Arrows etc are all based on. The `Maybe` data type defines two related



In a second well see how function application is different when something is a `Just a` versus a `Nothing` . First lets talk about Functors!

Functors

When a value is wrapped in a context, you cant apply a normal function to it:

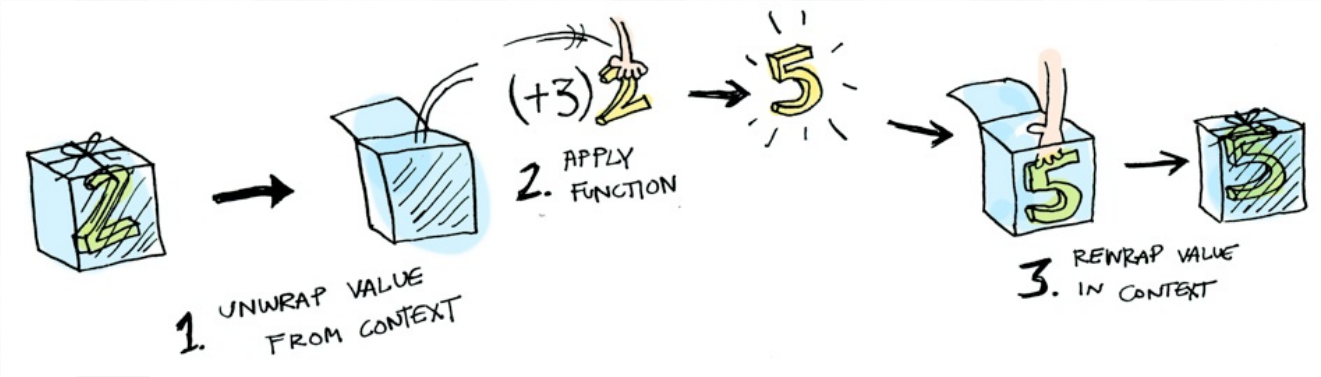


This is where `fmap` comes in. `fmap` is from the street, `fmap` is hip to contexts. `fmap` knows how to apply a function to a value with a context. You can use `fmap` on any type thats a `Functor` .
For example, suppose you want to apply `(+3)` to `Just 2` . Use `fmap` :

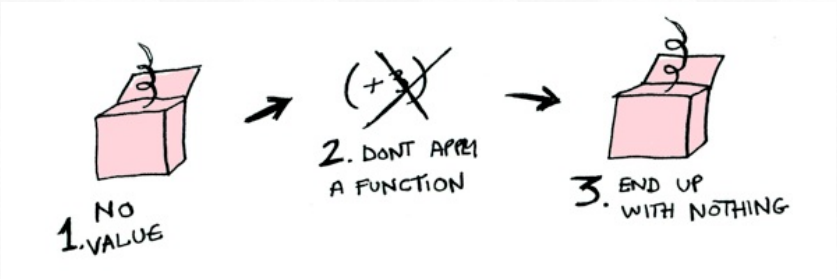
```
> fmap (+3) (Just 2)
Just 5
```

Heres what is happening behind the scenes:

Heres what is happening behind the scenes:



Bam! `fmap` shows us how its done!
So then youre like, alright `fmap`, please apply `(+3)` to a `Nothing`?




```
> fmap (+3) Nothing
Nothing
```

Like Morpheus in the Matrix, `fmap` knows just what to do; you start with `Nothing`, and you end up with `Nothing`! `fmap` is zen. So now youre all like, **just what is a Functor, exactly?** Well, a Functor is any data type that works with `fmap`. So `Maybe` is a functor. As well see soon, lists are functors too.
Now it makes sense why contexts exist. For example, heres how you work with a database record in a language without `Maybe`:

```
post = Post.find_by_id(1)
if post
  return post.title
else
  return nil
end
```

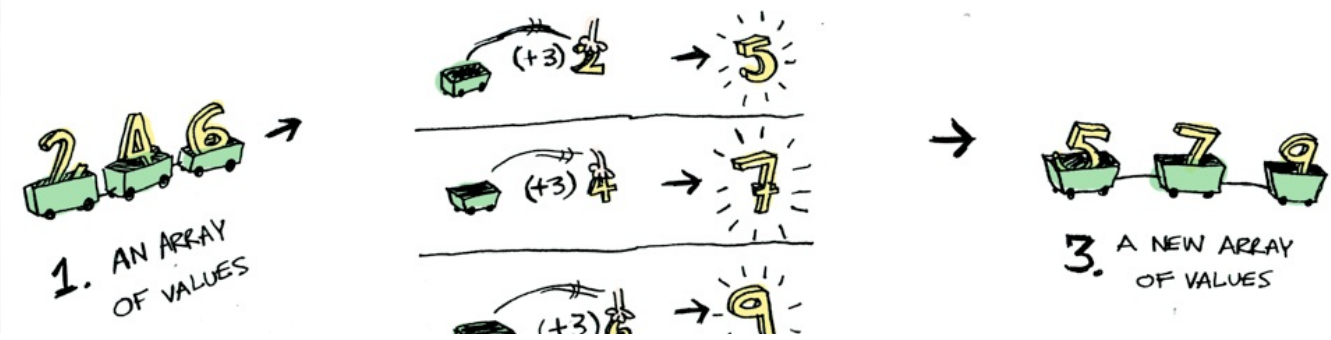
But in Haskell:

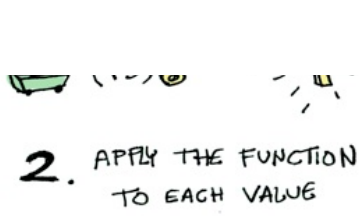
```
fmap (getPostTitle) (findPost 1)
```

If `findPost` returns a post, we will get the title with `getPostTitle`. If it returns `Nothing`, we will return `Nothing`!
Pretty neat, huh? `<$>` is the infix version of `fmap`, so you will often see this instead:

```
getPostTitle <$> (findPost 1)
```

Heres another example: what happens when you apply a function to a list?

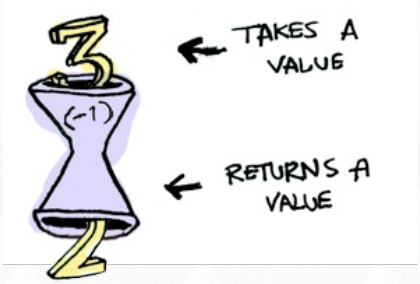




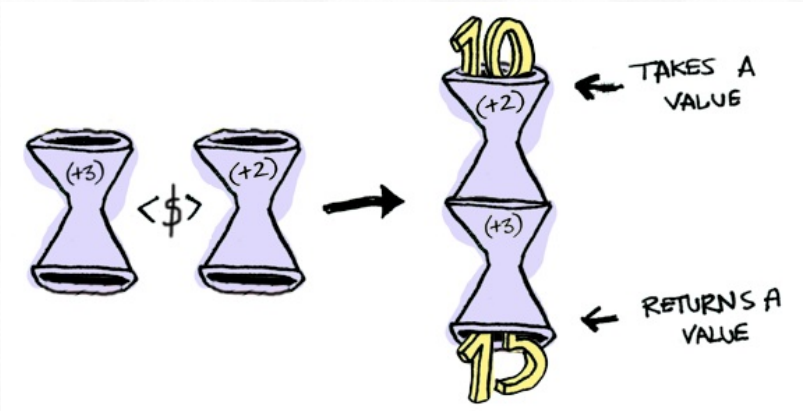
Lists are just another context that makes `fmap` apply the function differently!
Okay, okay, one last example: what happens when you apply a function to another function?

```
fmap (+3) (+1)
```

Heres a function:



Heres a function applied to another function:



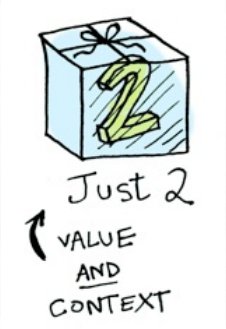
The result is just another function!

```
> import Control.Applicative
> let foo = (+3) <$> (+2)
> foo 10
15
```

Its just function composition! So, `f <$> g == f . g` !
Note: So far we have been treating the context like a box that holds a value. But sometimes the box analogy wears a little thin, like in this example. Just keep that in mind: boxes are useful mental pictures, but sometimes you dont have a box. Sometimes your box is a function.

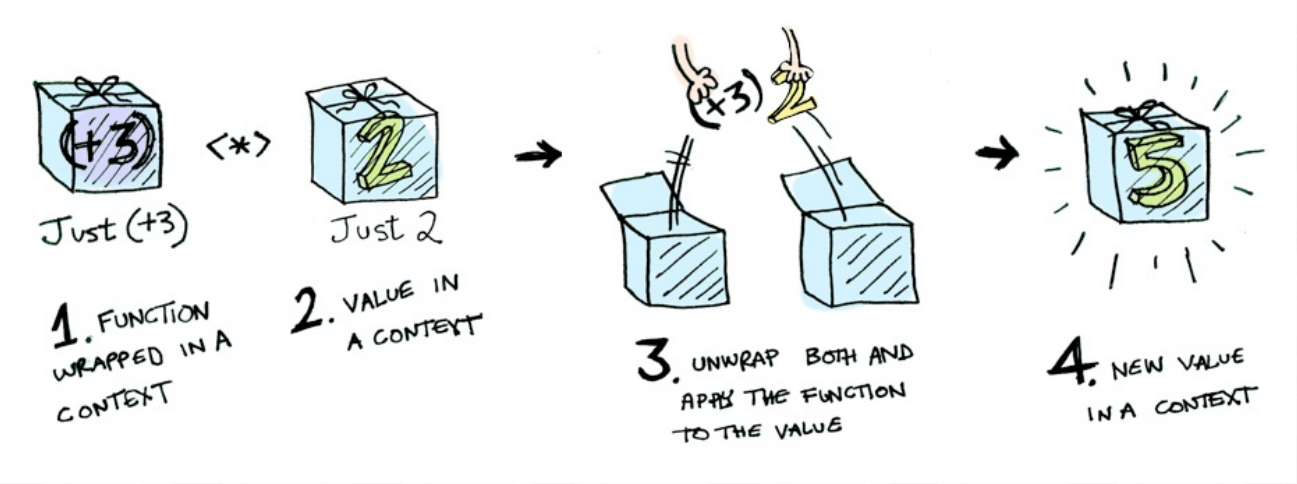
Applicatives

Applicatives take it to the next level. With an applicative, our values are wrapped in a context, just like Functors:



But our functions are wrapped in a context too!

Yeah. Let that sink in. Applicatives dont kid around. `Control.Applicative` defines `<*>`, which knows how to apply a function *wrapped in a context* to a value *wrapped in a context*:

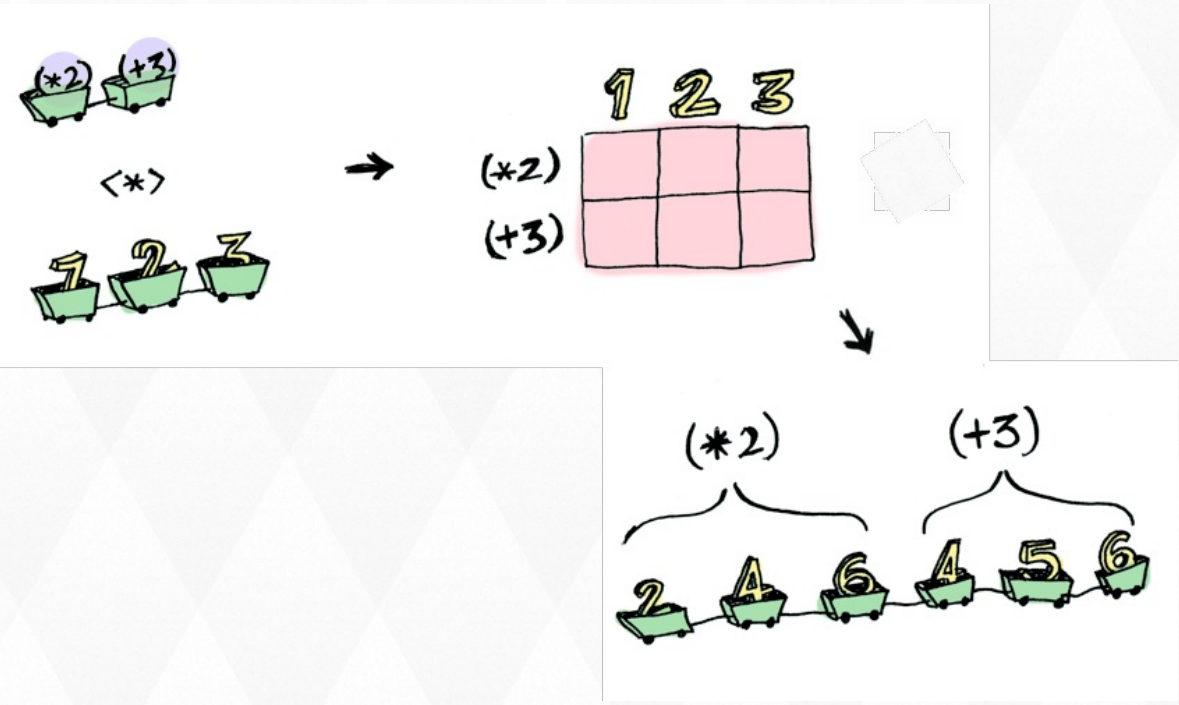


i.e:

```
Just (+3) <*> Just 2 == Just 5
```

Using `<*>` can lead to some interesting situations. For example:

```
> [(*2), (+3)] <*> [1, 2, 3]
[2, 4, 6, 4, 5, 6]
```



Heres something you can do with Applicatives that you cant do with Functors. How do you apply a function that takes two arguments to two wrapped values?

```
> (+) <$> (Just 5)
Just (+5)
> Just (+5) <$> (Just 4)
ERROR ??? WHAT DOES THIS EVEN MEAN WHY IS THE FUNCTION WRAPPED IN A JUST
```

Applicatives:

Applicatives:

```
> (+) <$> (Just 5)
Just (+5)
> Just (+5) <*> (Just 3)
Just 8
```

`Applicative` pushes `Functor` aside. Big boys can use functions with any number of arguments, it says. Armed `<$>` and `<*>`, I can take any function that expects any number of unwrapped values. Then I pass it all wrapped values, and I get a wrapped value out! AHAHAHAHAH!

```
> (*) <$> Just 5 <*> Just 3
Just 15
```

An applicative watching a functor apply a function

And hey! Theres a function called `liftA2` that does the same thing:

```
> liftA2 (*) (Just 5) (Just 3)
Just 15
```

Monads

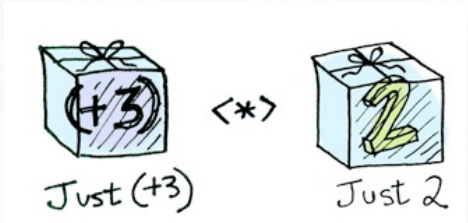
How to learn about Monads:

1. Get a PhD in computer science.
2. Throw it away because you dont need it for this section!

Monads add a new twist.

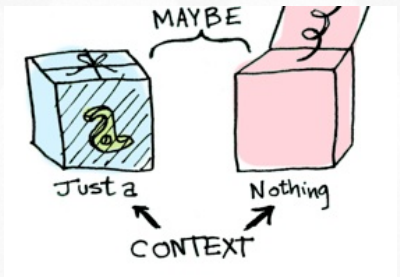
Functors apply a function to a wrapped value:

Applicatives apply a wrapped function to a wrapped value:



Monads apply a function **that returns a wrapped value** to a wrapped value. Monads have a function `>>=` (pronounced bind) to do this.

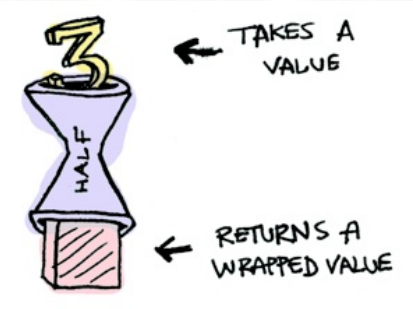
Lets see an example. Good ol `Maybe` is a monad:



Just a monad hanging out

Suppose `half` is a function that only works on even numbers:

```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```



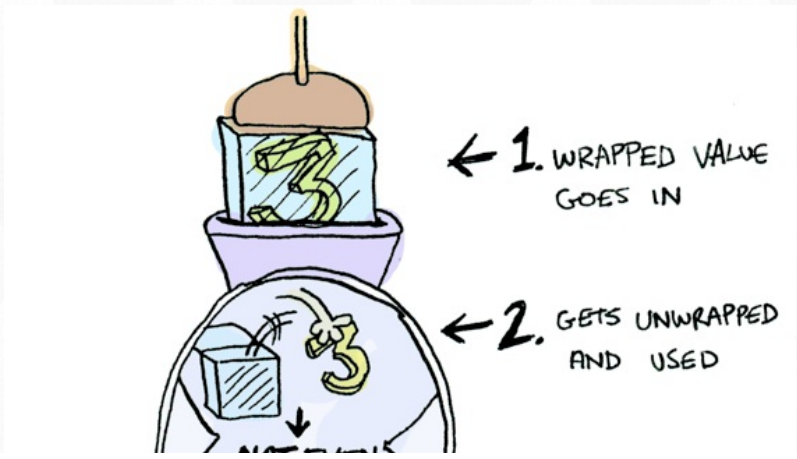
What if we feed it a wrapped value?

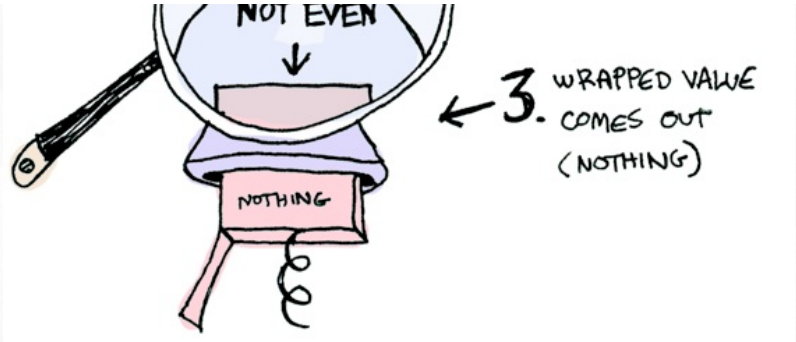
We need to use `>>=`, which will **force** our wrapped value into the function. Heres a photo of `>>=`:

Heres how it works:

```
> Just 3 >>= half
Nothing
> Just 4 >>= half
Just 2
> Nothing >>= half
Nothing
```

Whats happening inside?





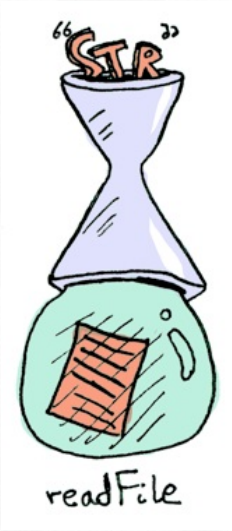
And if you pass in a `Nothing` its even simpler:

Cool stuff! Lets look at another example: the `IO` monad:

Specifically three functions. `getLine` takes no arguments and gets user input:

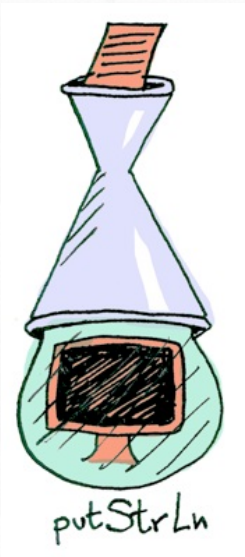

```
getLine :: IO String
```

`readFile` takes a string (a filename) and returns that files contents:



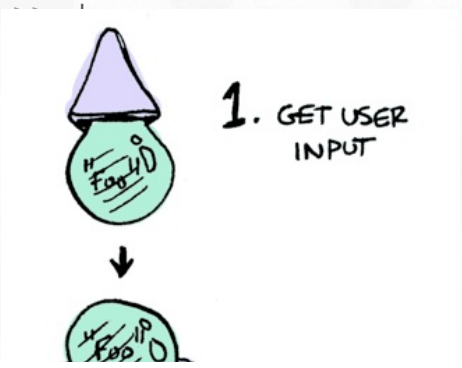

```
readFile :: FilePath -> IO String
```

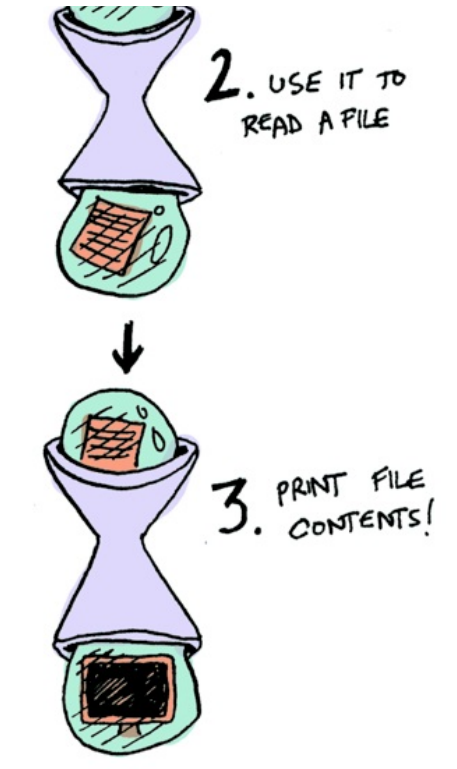
`putStrLn` takes a string and prints it:




```
putStrLn :: String -> IO ()
```

All three functions take a regular value (or no value) and return a wrapped value. We can chain all of these using



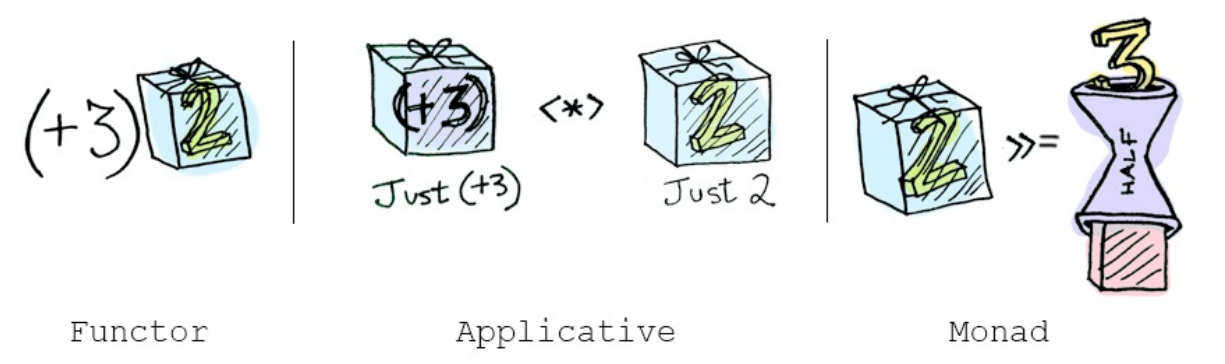



```
getLine >>= readFile >>= putStrLn
```

Aw yeah! We dont have time to waste, unwrapping and re-wrapping values in IO monads. `>>=` did that work for us. Haskell also provides us with some syntactical sugar for monads, called `do` notation:

```
foo = do
    filename <- getLine
    contents <- readFile filename
    putStrLn contents
```

Conclusion



- **functors:** you apply a function to a wrapped value using `fmap` or `<$>`
- **applicatives:** you apply a wrapped function to a wrapped value using `<*>` or `liftA`
- **monads:** you apply a function that returns a wrapped value, to a wrapped value using `>>=` or `liftM`

So, dear friend (I think we are friends by this point), I think we both agree that monads are easy and a SMART IDEA(tm). Now that youve wet your whistle on this guide, why not pull a Mel Gibson and grab the whole bottle. Check out LYAHs section on Monads. Theres a lot of things Ive glossed over because Miran does a great job going in-depth with this stuff.
